LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-211122

Proposed Babel/SIDL Changes to Support RMI

G. Kumfert, J. Leek

April 6, 2005

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Proposed Babel/SIDL Changes to Support RMI

A Working Document for Babel and Networking Developers
28 October 2004
(updated) 6 December 2004
(updated & released to public) 5 April 2005

Gary Kumfert and Jim Leek

Table of Contents

I. Goals

RMI support in Babel has two main goals: transparency & flexibility. The additional RMI feature should be transparent to existing Babelized code, allowing painless upgrade. The RMI capability should also be flexible enough to support a variety of RMI transport implementations.

The first goal's ideal would be for Babel users at some future date to simply upgrade to a RMI-enabled Babel release, regenerate files over their existing implementations, and find all their code is now able to be remotable without extensive modifications to their Impl files. The primary strategy for accomplishing the first goal is careful design and implementation of Babel generated code to minimize impact on user code.

The second goal's ideal would be for Babel users to plug in robust WebServices-like modules when accessing Babel objects across a WAN, and utilize faster binary protocol for accessing Babel objects across a LAN, or even different nodes in a leadership-class supercomputer --- without need to recompile their code. The primary strategy for accomplishing this second goal is to partner with appropriate parties to define an RMI API layer (in SIDL) such that various transport mechanisms can be "plugged-in."

II. Current Vision

The current vision for how RMI implementations (which are by nature generic libraries) interact with Babel code (which is generated) is sketched out roughly in this section. Section III will start enumerating forseeable changes to make this sketch viable. For the purposes of this discussion, we assume the following example:

```
package pkg version 1.0 {
    class cls {
        bool mthd ( in double idbl, out float oflt,
                    in string istr) throws Exception;
  }
}
```

A. Client Side Interactions

First a connection must be established between a stub, and a live object elsewhere. We will assume implementations (such as Proteus) may change the actual underlying protocol dynamically. From Babel's point of view, it cares not about the protocol across the wire, only an implementation (SIDL class) that implements the desired SIDL APIs.

1. Client Side Creation & Connection

Occurs when users use either static builtin methods "_create[Remote]()" or " _connect()"

on a Babel object. The former creates a new instance (and therefore only applies to SIDL classes), of a specific type on a named server using a named RMI implementation. The latter connects to an existing instance on named server based on its string argument. Connections may occur explicitly, and will often occur implicitly as objects are passed by reference in argument lists.

Regardless of the language binding, calls to _create[Remote]() will be delegated to the sidl.rmi.ProtocolFactory class implemented in the sidl runtime library[1].

```
sidl_rmi_InstanceHandle ih =
sidl_rmi_ProtocolFactory_createInstance( url, typeName );
```

The ProtocolFactory will in turn parse the URL, identify an implementation associated with the protocol portion of the URL, use sidl.Loader to instantiate it, and initialize it with a call to Connection.init(). The RMI library will implement the sidl.rmi.InstanceHandle interface and Babel stubs when connected to remote objects will store the handle to the sidl.rmi.InstanceHandle IOR in their d_data.

2. Client Side Method Invocation

When Stubs to remote objects are invoked, Babel generated code will dispatch through the EPV, but the EPV will be initialized specifically for RMI dispatch. Those Babel generated functions will always be in C, and look like

```
sidl_rmi_InstanceHandle = (sidl_rmi_InstanceHandle) d_data;
sidl_rmi_Invocation i =
    sidl_rmi_InstanceHandle_readyInvocation( c, "mthd");

sidl_rmi_Invocation_packDouble( i, "idbl" , 2.0 );
sidl_rmi_Invocation_packString( i, "istr", "Hello" );

sidl_rmi_Response r = sidl_rmi_Invocation_invokeMethod( i );

sidl_rmi_Response_unpackSerializable( r, "_ex", &ex );
if ( ex._not_nil() ) {
     /* handle exception*/
} else {
    sidl_rmi_Response_unpackBool( r, "_retval", &_retval);
    sidl_rmi_Response_unpackFloat( r, "oflt", &oflt );
}
```

1Note: sidl.rmi.InstanceHandle was called sidl.rmi.Connection in the initial draft. Objections were raised that the lifetime of the object spanned potentially multiple TCP/IP connections

Where the RMI libraries implement the following SIDL interfaces: sidl.rmi.InstanceHandle, sidl.rmi.Invocation, and sidl.rmi.Response.

3. Client Side Deletion or Disconnect

All Babel objects are reference counted and when reference count goes to zero, the object's destructor is invoked. However, Babel neither requires nor enforces that the reference count be known exactly at any one location. For instance, the reference count for an object in the IOR could be 2, but one of those could be from Python where Python has 5 internal references to the extension module. In this case, only when all 5 internal references are collected in Python would the IOR's reference count be decremented by one, and only when the IOR's reference count goes to zero would the instance be destroyed.

This paradigm carries over to RMI. The actual instance need only keep track of the number of remote locations that access it, not the exact number of stubs at each location. Only when the last stub referencing a remote object is reclaimed should a deleteReference call be made to the server. In a sense, the InstanceHandle's reference count serves double-duty as the local reference count for the remote object. This means that remote stubs will maintain their own reference count, and only the last call to deleteReference will actually close the connection. Whether that closed connection constitutes a need to call _dtor() on the instance is the duty of the server side.

B. Server Side Interactions

Whether the object pre-exists and is posted, or is created on demand by some kind of server, eventually some general library that handles this needs to interact with Babel generated code. This section attempts to sketch out the interactions.

1. Server Side Creation & Connection

Babel already has a mechanism to create instances based on typeName via the sidl.Loader. It must also provide an anaologous mechanism to both (a) access instances based on an instanceName and (b) to execute methods on instances by methodName We propose adding a sidl.rmi.InstanceRegistry singleton class whose methods are all static and protected by read/write mutexes. All instances to be remoted will be added to the registry. [Q: When to remove objects from the registry?]

2. Server Side Method Invocation

It is assumed that when a remote method is invoked, the RMI library will receive the self-describing stream of data that directs it to dispatch to Babel-generated code. Therefore, the Babel objects that the RMI library interfaces with are defined in SIDL to permit RMI libraries written in multiple languages. The RMI library may multiplex streams to instances on its own, or rely on the InstanceRegistry to get unique handles to instances. The RMI library on the server-side will then perform the following sequence:

```
sidl_io_Deserializer inArgs = ... ;
sidl_io_Serializer outArgs = ...;

sidl_BaseClass bc =
      sidl_InstanceRegistry_getInstance ( "instanceID" );
sidl_BaseClass__exec( bc, "mthd", inArgs, outArgs );
```

And then Babel' s"_exec()" method will dispatch to generated code specific to the named method and instance.

```
/* in pkg_cls_mthd__rserver(...) */
double idbl;
float oflt;
char * istr;
SIDL_bool _retval;
SIDL_BaseClass _ex;

sidl_io_Deserializer_unpackDouble( inArgs, "idbl", &idbl );
sidl_io_Deserializer_unpackString( inArgs, "istr", &istr );

_retval = pkg_cls_mthd( self, idbl, oflt, istr, &_ex);

if ( _ex != NULL ) {
    /* register exception and return a handle to it */
} else {
    sidl_io_Serializer_packBool( outArgs, "_retval", _retval );
    sidl_io_Serializer_packFloat( outArgs, "oflt", oflt );
}
```

For cases where argument ordering is all that is required, the string names may be ignored by the RMI library. For cases where the stream is self-describing and arguments may arrive out of order, the RMI library implementor will have to queue up the data for later query. The RMI library will have to implement the sidl.io.Serializer and sidl.io.Deserializer interfaces.

3. Server Side Deletion & Disconnect

[Q: How to properly reclaim instances when an outstanding reference exists in the in the InstanceRegistry?]

III. Planned Modifications to Support Vision

A. New Builtin Methods

Builtin methods start with a leading underscore (which is not expressible in SIDL) and get generated by the backends for each language. They are never virtual functions.

1. _create[Remote](in string url)

Similar to _create(), this is behaves like a static final method. It differs in that the url argument is parsed to determine the RMI library, remote server, and port number to use in creating a new remote instance. Exists only for classes and may return an empty stub if the creation fails. [Q: How to communicate why it failed?]

2. _connect(in string url, in string instanceID)

Instead of creating a new remote instance, this one creates a new connection to an existing remote object. The object's reference count will be incremented for the duration of the connection. Interfaces will also be able to connect to remote instances.

3. _exec(in sidl_BaseClass self, in string methName, in sidl_io_Deserializer inArgs, in sidl_io_Serializer outArgs)

This method would provide a fundamentally new capability to Babel objects in that their methods could be invoked by name. The method name would almost certainly have to be the long form since overloading is impossible to support. Essentially, the implementation would have to cast to the implementation type, then look up the appropriate function pointer by string name, then implement a C function for each method that unpacks the inArgs, executes the normal EPV entry, and packs the outArgs.

B. Additions to sidl.sidl & sidl runtime library

1. sidl.adt.TypedMap class

Almost identical to the decaf.TypeMap implementation. The only problem is that the current implementation is in C++ making extensive use of STL map<>. [Q: should we support nested TypedMaps? CCA spec doesn't and I never understood why]. Would also make sense for the TypedMap to implement Serializer and Deserializer interfaces for ease of implementing RMI libraries for Babel.

2. sidl.rmi.InstanceRegistry class (singleton)

Basically generates unique names for each registered instance. And serves up instances by name. [Q: proper instance removal is an unanswered question.]

```
class InstanceRegistry {

  /**
   * register an instance of a class
   *  the registry will return a string guaranteed to be unique
   *  for the lifetime of the process
   */
  static string registerInstance( in sidl.BaseClass instance );

  /**
   * returns a handle to the class based on the unique string
   */
  static sidl.BaseClass getInstance( in string instanceID );
}
```

3. sidl.io.Serializer interface

Can be implemented by RMI libraries, IO libraries, maybe even checkpoint/restart and process migration. [Note: "pack" is an intrinsic function in F90. The F90 bindings will need to address the problem that some F90 compilers treat intrinsics as reserved words.]

```
interface Serializer {
  void pack[Bool]( in string key, in bool value ) throws IOException ;
  void pack[Char]( in string key, in char value ) throws IOException ;
  void pack[Int]( in string key, in int value ) throws IOException ;
  void pack[Long]( in string key, in long value ) throws IOException ;
  void pack[Float]( in string key, in float value ) throws IOException ;
  void pack[Double]( in string key, in double value ) throws IOException ;
  void pack[Fcomplex]( in string key, in fcomplex value ) throws IOException ;
  void pack[Dcomplex]( in string key, in dcomplex value ) throws IOException ;
  void pack[String]( in string key, in string value ) throws IOException ;
  void pack[Serializable]( in string key, in Serializable value )
       throws IOException;

  /* similar for packing arrays of values */
  ...
}
```

4. sidl.io.Deserializer interface

Compliments features from Serializer.[Note: "unpack" is an intrinsic function in F90. The F90 bindings will need to address the problem that some F90 compilers treat intrinsics as reserved words.]

```
interface Deserializer {
  void unpack[Bool]( in string key, out bool value ) throws IOException ;
  void unpack[Char]( in string key, out char value ) throws IOException ;
  void unpack[Int]( in string key, out int value ) throws IOException ;
  void unpack[Long]( in string key, out long value ) throws IOException ;
  void unpack[Float]( in string key, out float value ) throws IOException ;
  void unpack[Double]( in string key, out double value ) throws IOException ;
```

```
    void unpack[Fcomplex]( in string key, out fcomplex value ) throws IOException ;
    void unpack[Dcomplex]( in string key, out dcomplex value ) throws IOException ;
    void unpack[String]( in string key, out string value ) throws IOException ;
    void unpack[Serializable]( in string key, out Serializable value )
             throws IOException;

    /* similar for unpacking arrays of values */
  }
```

5. sidl.io.Serializable interface

This would be implemented by classes that can pack and unpack themselves to a Serializer/Deserializer.

```
 interface Serializeable {
   void pack( in Serializer ser );
   void unpack( in Deserializer des );
 }
```

6. sidl.rmi.ProtocolFactory class (singleton)

Serves up RMI libraries (e.g. proteus) using sidl.Loader based on a string that is found in the URL and almost definitely not the same as the class name. Note that all these methods should throw NetworkExceptions.

```
class ProtocolFactory {
  /**
   * Associate a particular prefix in the URL to a typeName
   * <code>sidl.Loader</code> can find.  The actual type is
   *  expected to implement <code>sidl.rmi.InstanceHandle</code>
   * Return true iff the addition is successful (no collisions
   * allowed)
   */
  static bool addProtocol( in string prefix, in string typeName )
      throws NetworkException;

  /**
   * Return the typeName associated with a particular prefix.
   * Return empty string if the prefix
   */
  static string getProtocol( in string prefix )
      throws NetworkException;

  /**
   * Remove a protocol from the active list.
   */
  static bool deleteProtocol( in string prefix )
    throws NetworkException;

  /**
```

```
     * Create a new, initialized connection based on the input
     * string.  Return ni if protocol unknown or Connection.init()
     *  failed.
     */
    static InstanceHandle createInstance( in string url,
                                          in string typeName );
      throws NetworkException;


     /**
      * Create an new connection linked to an already existing
      * object on a remote
      * server.  The server and port number are in the url, the
      * objectID is the unique ID
      * of the remote object in the remote instance registry.
      * Return nil if protocol unknown or InstanceHandle.init()
      * failed.
      */
     static InstanceHandle connectInstance( in string url, in string
        typeName, in string objectID ) throws NetworkException;

  }
```

7. sidl.rmi.InstanceHandle interface

Implemented by the RMI library. (Was called "Connection" in earlier versions, but name was changed in acknowledgement that this object may persist over multiple network connections. This interface extends Serializable so that exceptions and remote object references can be exchanged.

```
  interface InstanceHandle extends sidl.io.Serializable {

    /**
     * initialize a connection (intended for use by the
     * ProtocolFactory)
     */
    bool init[Create]( in string protocol, in string server,
                       in int port, in string typeName )
        throws NetworkException;

    /**
     * initialize a connection (intended for use by the
     * ProtocolFactory)
     */
    bool init[Connect]( in string protocol, in string server,
                        in int port, in string typeName.
                         in string objectID )
        throws NetworkException;
    /** return the name of the protocol */
    string getProtocol() throws NetworkException;

    /** return the name of the server */
    string getServerName() throws NetworkException;

    /** return the port number on the server */
    int getPort() throws NetworkException;

    /** return the objectID */
    string getObjectID() throws NetworkException;

    /** create a handle to invoke a named method */
    Invocation createInvocation( in string methodName )
        throws NetworkException;

    /** closes the connection (called be destructor, if not done
     * explicitly) returns true if successful, false otherwise
     * (including subsequent calls)
     */
    bool close() throws NetworkException;
  }
```

8. sidl.rmi.Invocation interface

Implemented by the RMI library. Exists for the duration of a single method invocation.

```
  interface Invocation extends sidl.io.Serializer {

    /**
     * this method may be called only once at the end of the
     *  object's lifetime
     */
    Response invokeMethod() throws NetworkException;
  }
```

9. sidl.rmi.Response interface

Implemented by the RMI library. Encapsulates the response of a single method invocation.

```
  interface Response extends sidl.io.Deserializer {

    /** returns true iff RMI hasn't timed out */
    bool timedOut() throws NetworkException;

    /** if returns null, then safe to unpack arguments */
    sidl.BaseException getExceptionThrown() throws NetworkException;
```

```
        /** signal that all is complete */
        bool done() throws NetworkException;
      }
```

C. Modifications to Babel Code Generation

For the in-process case: Babel always used the Stubs as the main accessor of libraries and the IOR as the point of providing. All language bindings currently have Stubs at least partially implemented in C/C++. IORs are always implemented in C. For minimial impact to our customers, we intend to generate the extra support code for RMI in the Stub and IOR C files. The new builtin's in Section A will need to be added to each binding on a case-by-case basis.

As the code is general across all language bindings, the actual Java source to generate code is being developed in babel-x.x.x/compiler/gov/llnl/babel/backend/rmi/*.java.

As of December 6, 2004 there has already be significant restructuring of generated source to better separate Stubs and IOR source files, and much of the Stub infrastructure has been drafted out. Work on IOR extensions is expected to commence in January 05.

D. Non-blocking RMI

There is significant interest in non-blocking RMI to interleave the computation with communication. Evidence that this technique is critical to achieving performance in SPMD programming is legion. Work on implementing this in RMI is not. Noteworthy is Kate Kehey's graduate work with PARDIS at Indiana University. Unfortunately, her work does not consider the possibility of the exceptions being thrown from a non-blocking RMI.

Expect future revisions of this document to add more design informaiton here.